Atanas Delevski

ECE 559 Midterm #1

10/12/2020

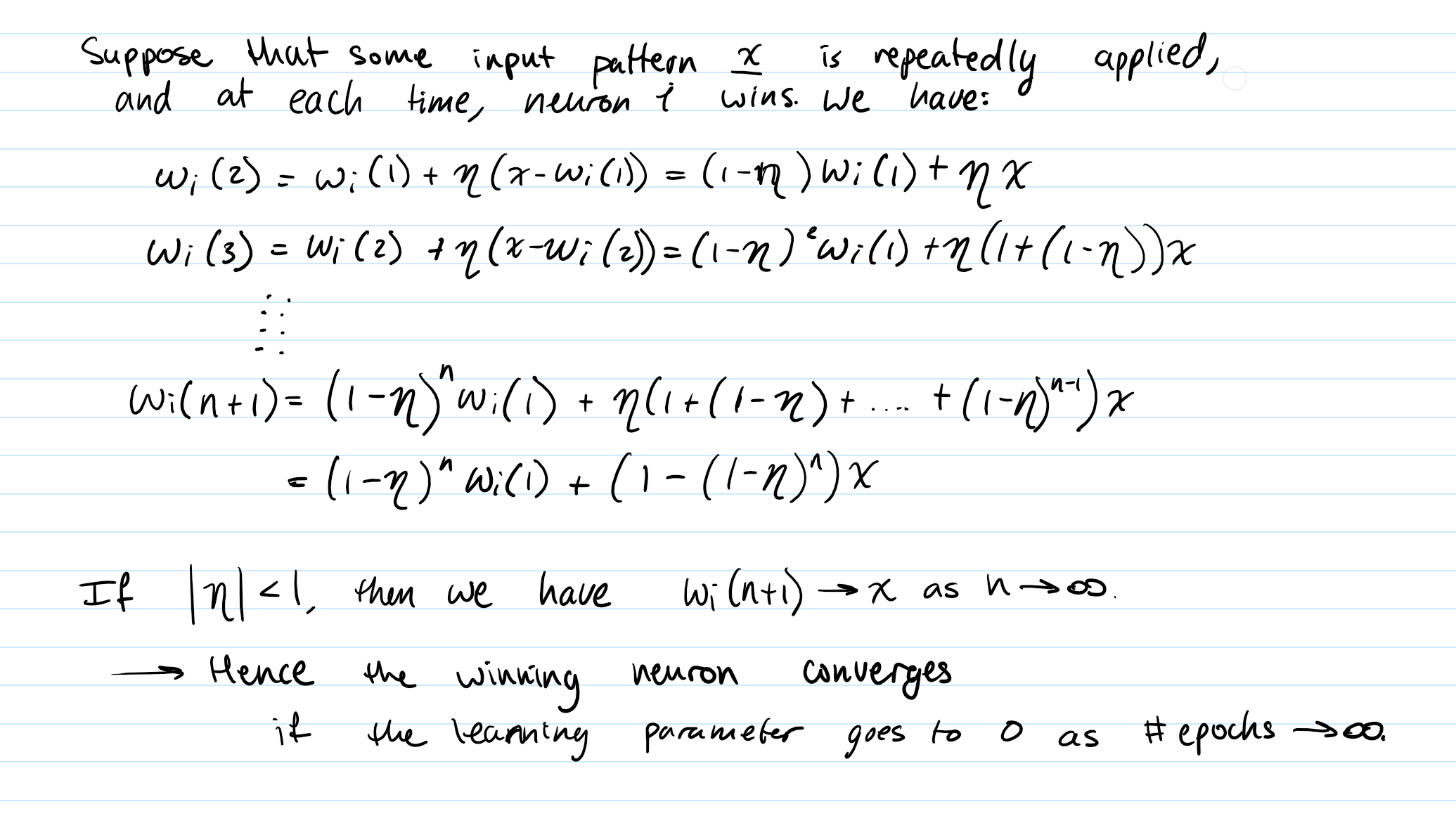
**Question 1:**

Part 2:

Since the image is 512 by 512, and each pixel has a value between [0 and 255], the number of bits needed to represent the image are 8\*512\*512 = 2,097,152 bits of information. We need 8 bits to represent the possibilities of [0 to 255] (2^8) and 512\*512 is the number of pixels we have.

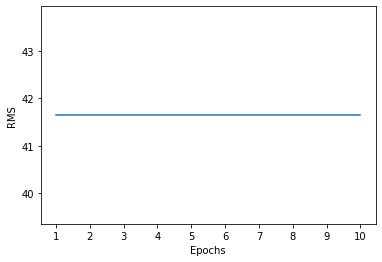
Part 6:

Yes, the weights converge. Given the proof below, our weights will also converge since the weight updating process remains the same even if the method to find the winning neuron differs.

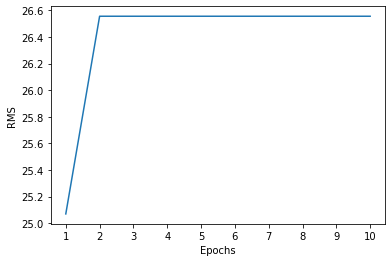


Part 8:

For d=4, the matrix is 128 by 128, and each submatrix is 4 by 4. This time we only have 8 possible submatrices. So the number of bits needed to represent the image are 8\*8\*16 + 3\*128\*128 = 1024 + 49152 = 50176 bits of information. Where 8\*8\*16 accounts for the 8 bits of information needed for [0-255], the 8 possible submatrices, and the 16 values each submatrix has, and then the 3\*128\*128 accounts for the 3 bits needed to tell which submatrix you need for the 128 by 128 submatrices of the compressed image out of the 8 you have available (2^3).

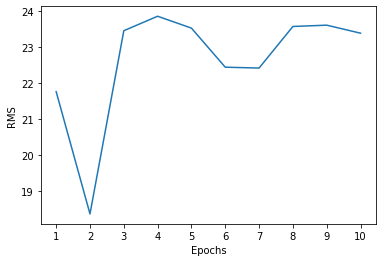
Part 9:

K = 2:



K = 8:





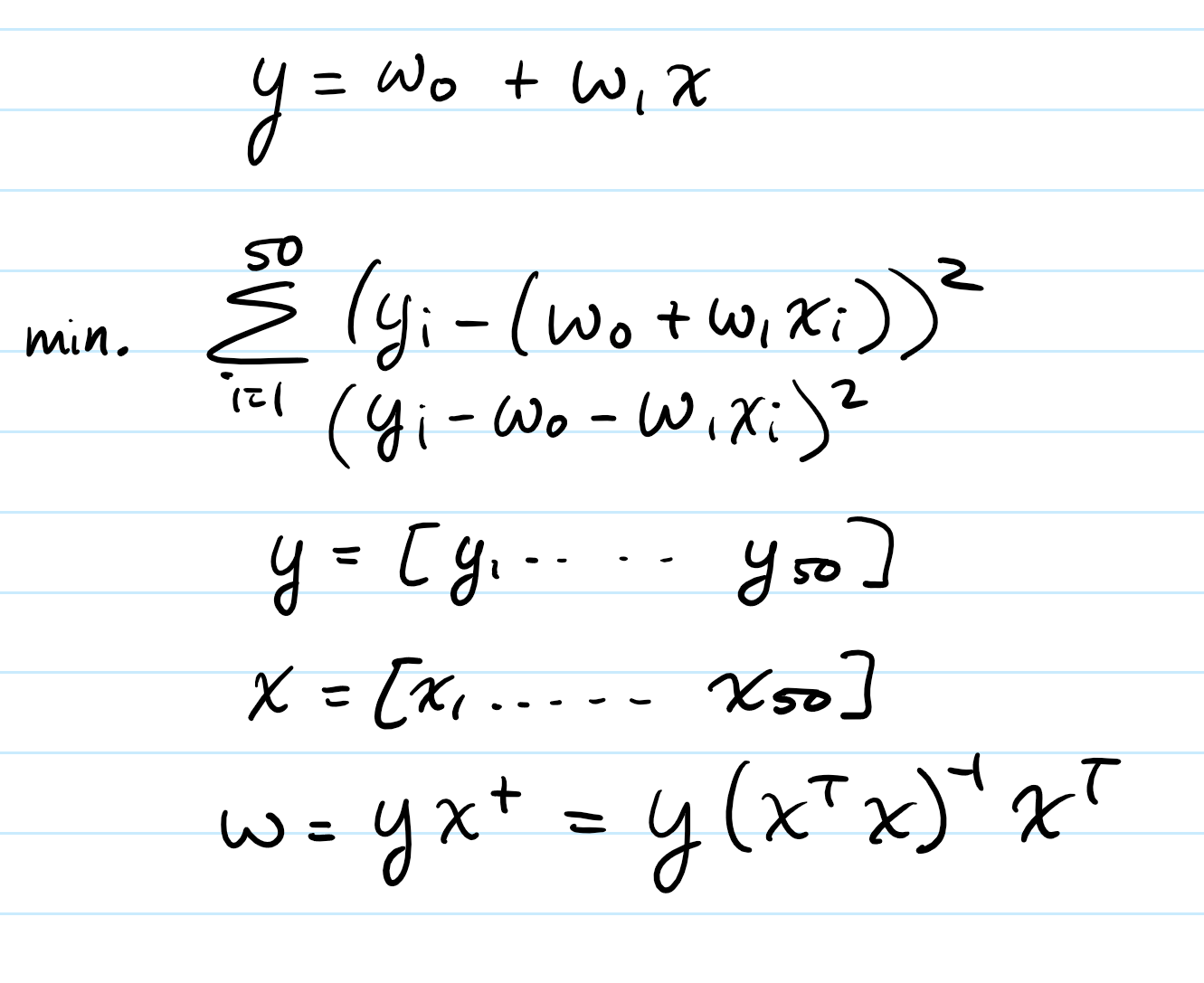
K = 32:

Part 10:

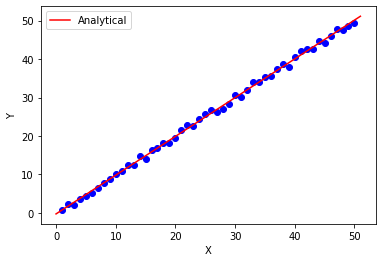
The competitive learning method achieves compression while remaining faithful to the original image, because it guarantees to train K neurons to represent the most common pixel intensity combinations. Therefore, the more neurons you have, the better job it will do. But no matter the K, it will always remain faithful by relying on the pixel intensity probability distribution of the original image.

**Question 2:**

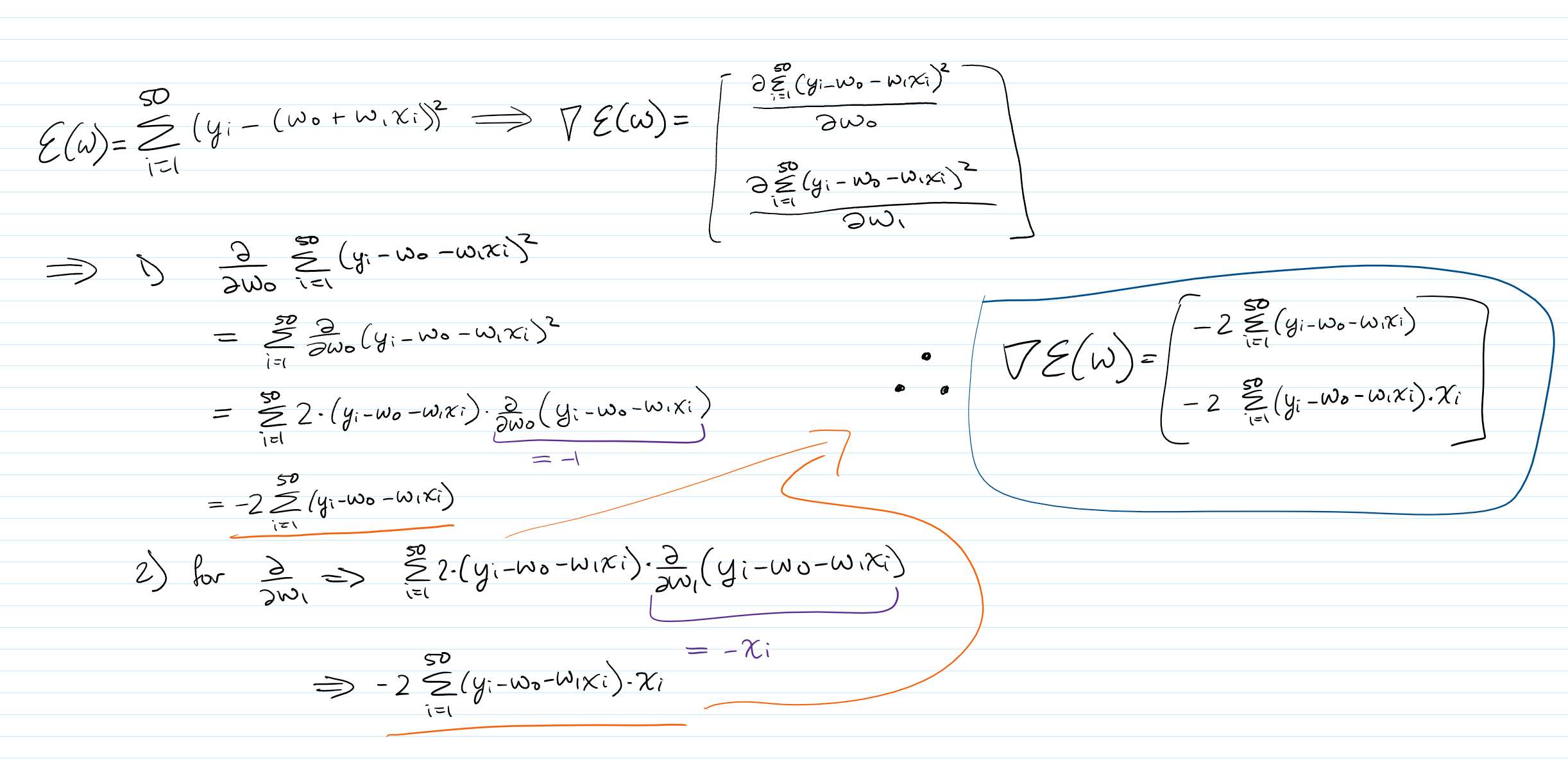
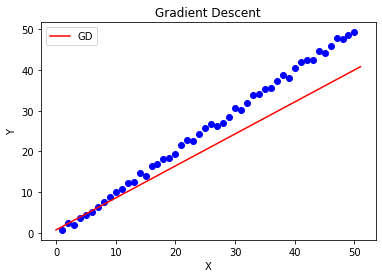
Part 2:



Part 3: ANALYTICAL



Part 4:



Part 5:

Part 6:

The LLS method fits the data really perfectly, but the GD method does not. Instead, it skews off to the right, no matter how many epochs I train it for. I think this is due to bad weight initialization. Either that, or there were not enough iterations of the gradient descent.

CODE (QUESTION 1):

##### MIDTERM 1, PROBLEM 1 - ATANAS DELEVSKI #######

from matplotlib import pyplot as plt

import numpy as np

np.random.seed(1)

######### DECLARING VARIABLES ##########

d = 4

K = 8

num\_epochs = 10

learning\_rate = 0.25

######################################

###### INITIAL IMAGE HANDLING ########

pre = plt.imread('barbara.png')

imglen = len(pre[0])

img = np.zeros((imglen, imglen))

for i in range(imglen):

for j in range(imglen):

img[i][j] = int(pre[i][j]\*255) # increasing values to [0,255]

# plt.imshow(img, cmap='Greys\_r')

######################################

#### CREATING THE "A" MATRIX (TRAIN SET) & REVERSAL FUNCTION ######

n = int(imglen/d)

A = np.zeros((n, n, d, d))

k\_index = 0

for k in range(0, n):

l\_index = 0

for l in range(0, n):

A[k][l] = img[k\_index:k\_index+d, l\_index:l\_index+d]

l\_index += d

k\_index += d

def reverse(matrix):

size = len(matrix[0])

B = np.zeros((size\*d, size\*d))

k\_index = 0

for k in range(0, n):

l\_index = 0

for l in range(0, n):

B[k\_index:k\_index+d, l\_index:l\_index+d] = matrix[k][l]

l\_index += d

k\_index += d

return B

################################################

######## CLASS & FUNCTIONS ############

class Competitive\_Learning\_Network(object):

def \_\_init\_\_(self, num\_of\_neurons, neuron\_size):

self.num\_of\_neurons = num\_of\_neurons

self.neuron\_size = neuron\_size

self.neuron\_weights = 255 \* np.random.rand(num\_of\_neurons, neuron\_size, neuron\_size)

def train(self, train\_set, eta, num\_of\_epochs):

self.num\_of\_epochs = num\_of\_epochs

self.rms\_values = []

winners = np.zeros(K).astype(int)

for epoch in range(self.num\_of\_epochs):

# print(f"Starting Epoch #{epoch+1}")

for i in range(n):

for j in range(n):

values = []

for k in range(K):

temp = train\_set[i][j]-self.neuron\_weights[k]

value = sum(sum((temp)\*(temp)))

values.append(value)

index = values.index(min(values))

winners[index] = 1

self.neuron\_weights[index] = (1-eta)\*self.neuron\_weights[index] + eta\*train\_set[i][j]

for i in range(K):

if winners[i] != 1:

# print(f"Neuron {i} never won, reinitializing..")

self.neuron\_weights[i] = train\_set[0][0]

# print(f"Ending Epoch #{epoch+1}")

rms, a\_prime\_reversed = self.test(train\_set)

self.rms\_values.append(rms)

print(f"RMS after epoch #{epoch+1}: {rms}")

return a\_prime\_reversed

# print(f"The Resutling RMS values for each epoch are: {self.rms\_values}")

def test(self, train\_set):

A\_prime = np.zeros((n, n, d, d))

rms = 0

for i in range(n):

for j in range(n):

values = []

for k in range(K):

temp = train\_set[i][j]-self.neuron\_weights[k]

value = sum(sum((temp)\*(temp)))

values.append(value)

index = values.index(min(values))

A\_prime[i][j] = self.neuron\_weights[index]

a\_prime\_reversed = reverse(A\_prime)

train\_set\_reversed = reverse(train\_set)

temp = a\_prime\_reversed-train\_set\_reversed

rms = (1/512)\*np.sqrt(sum(sum((temp)\*temp)))

return rms, a\_prime\_reversed

#################################################

########### USAGE ################

network = Competitive\_Learning\_Network(K, d)

compressed = network.train(A, learning\_rate, num\_epochs)

plt.imshow(compressed, cmap="Greys\_r")

plt.show()

##################################

######### RMS/EPOCH GRAPH #############

# epochs = np.arange(1, network.num\_of\_epochs+1)

# plt.plot(epochs, network.rms\_values)

# plt.xlabel('Epochs')

# plt.xticks(np.arange(1, network.num\_of\_epochs+1, step=1))

# plt.ylabel('RMS')

# plt.show()

#######################################

CODE (QUESTION 2):

########### MIDTERM QUESTION 2 - ATANAS DELEVSKI ###########

import numpy as np

import matplotlib.pyplot as plt

np.random.seed(1)

########## DECLARING VARIABLES ###########

u = 2 \* np.random.rand(50) - 1

n = 50

X = [i for i in range(1, n+1)]

Y = [i + u[i-1] for i in range(1, n+1)]

##########################################

################ PART 1 ##################

num = 0

den = 0

for i in range(n):

num = num + (X[i] - np.mean(X)) \* (Y[i] - np.mean(Y))

den = den + np.power(X[i] - np.mean(X), 2)

x = np.linspace(0, n+1, 1000)

y = (np.mean(Y) - ((num / den) \* np.mean(X)) + (num / den)\*x)

plt.plot(x, y, color='r', label='LLS')

plt.scatter(X, Y, color='b')

plt.xlabel('X')

plt.ylabel('Y')

plt.title("Analytical")

plt.legend()

plt.show()

##########################################

####### DECLARING VARIABLES #2 ###########

x = np.array(X)

y = np.array(Y)

w = np.array([0, 1])

lr = 0.01

epochs = 10

##########################################

########### FUNCTIONS ########################

def our\_sum(x, y, w):

sum = 0

for i in range(50):

sum += (y[i] - (w[0] + w[1]\*x[i]))\*\*2

return sum

def gradient(x, y, w):

g = np.zeros(2)

for i in range(n):

g[0] += (y[i]-w[0]-w[1]\*x[i])

g[1] += (y[i]-w[0]-w[1]\*x[i])\*x[i]

return -2\*g

def update\_weights(epochs, lr, x, y, w):

for i in range(epochs):

g = gradient(x, y, w)

new\_w = w - lr\*g

return new\_w

def get\_new\_y(x, y, new\_w):

new\_y = np.zeros(n)

for i in range(n):

new\_y[i] = new\_w[0] + new\_w[1]\*x[i]

return new\_y

############################################

######## USAGE ###################

new\_w = update\_weights(epochs, lr, x, y, w)

new\_y = get\_new\_y(x, y, new\_w)

##################################

########### GRAPH 2 ###############

x = np.linspace(0, n+1, 50)

plt.plot(x, new\_y, color='r', label='GD')

plt.scatter(X, Y, color='b')

plt.xlabel('X')

plt.ylabel('Y')

plt.title("Gradient Descent")

plt.legend()

plt.show()

################################